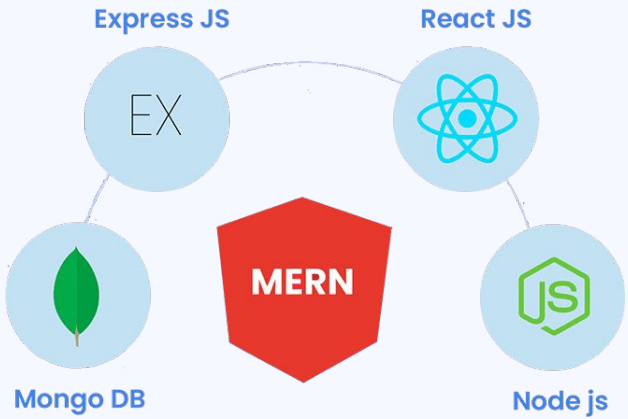


Full Stack Development Using MERN

Day 03 - 05 May 2024

Lecturer Details:

Prameeth Madhuwantha
bap@ucsc.cmb.ac.lk
0772888098

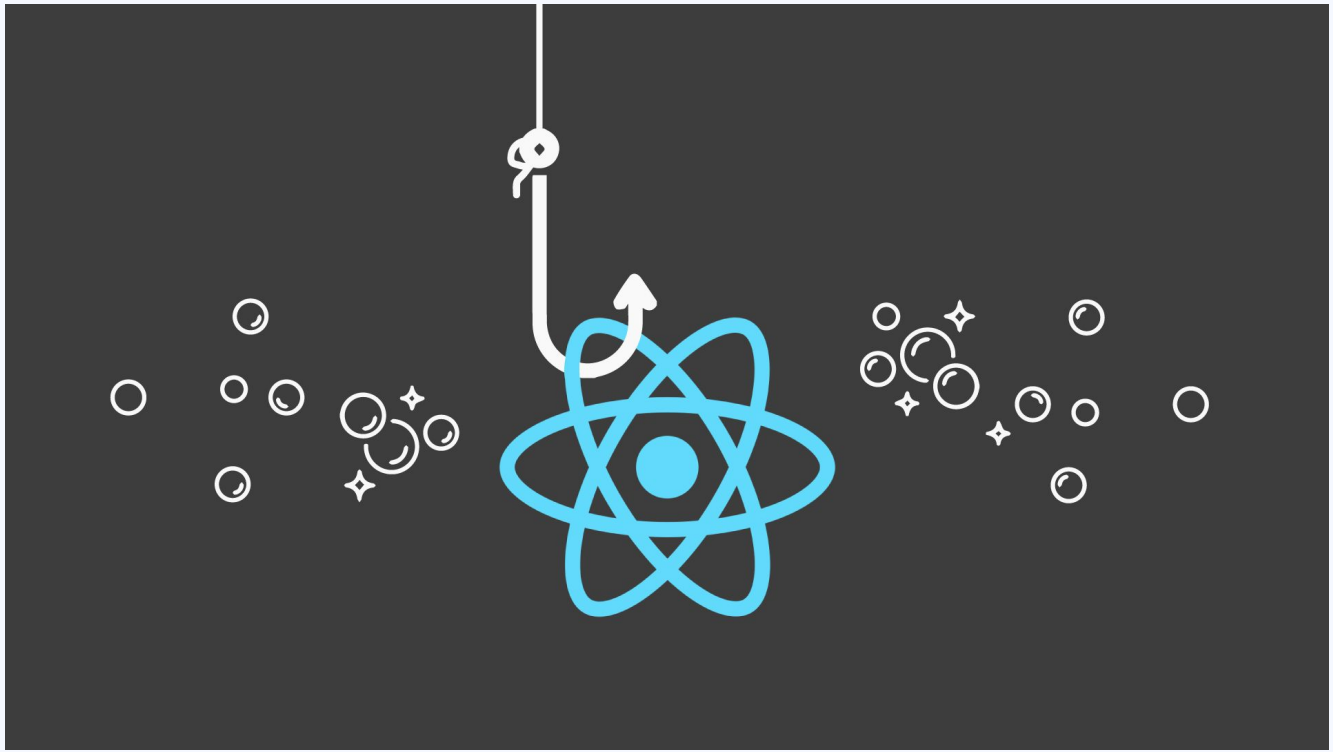
Chanaka Wickramasinghe
cmw@ucsc.cmb.ac.lk
0771570227

Anushka Vithanage
dad@ucsc.cmb.ac.lk
071 527 9016

Lecture 03

Hooks, Form Validations and React Routing with React.js

React Hooks



Hooks

A hook in React is a special kind of function that allows you to "hook into" React features from within function components

- useState
- useEffect
- useContext
- useReducer
- useCallback
- useMemo
- useRef
- useLayoutEffect

State

State refers to the local state of a component. It is data that dictates how a component renders and behaves. Unlike props, the state is changeable, but there's a proper way to change it.

Initialization: State is often initialized using the useState hook in functional components.

Changing State: In functional components, you use the setState function returned by the useState hook.

State

Rerender: Whenever the state changes, the component re-renders. It's one of the reasons why React components can be dynamic.

Local and Encapsulated: Each component has its own state. It's not accessible to any component other than the one that owns and sets it.

useState

- Allows functional components to maintain local state
- useState hook, returns an array with two elements:
  - The current state value
  - A function to update that state value.
    - Eg:

```
const [count, setCount] = useState(0);
```

- count is the first item, which is the current state value. It is initialized to 0.
- setCount is the second item, which is a function that you can use to update the value of count.

State - Example

```jsx
import React, { useState } from 'react'

export default function UseState() {

  const [count, setCount] = useState(0);

  return (
    <div>
        <h1>UseState Example - Counter</h1>
        <p>You clicked {count} times</p>
        <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  )
}
```

useState Example

2. Strings with useState

```jsx
export default function StringsWithUseState() {
    const [text, setText] = useState("Hello, World!");

    return (
        <div>
            <p>{text}</p>
            <button onClick={() => setText(text.split('').reverse().join(''))}>
                Reverse Text
            </button>
            <button onClick={() => setText(text.toUpperCase())}>
                Uppercase
            </button>
            <button onClick={() => setText(text.toLowerCase())}>
                Lowercase
            </button>
        </div>
    );
}
```

useState Example

3. **Arrays with useState**

```jsx
const [items, setItems] = useState(['Apple', 'Banana', 'Cherry']);

const addItem = () => setItems([...items, 'Date']);

const removeItem = (index) => {
  setItems(items.filter((_, i) => i !== index));
};

const updateItem = (index) => {
  // Assuming you want to update item in some manner
  const newItems = [...items];
  newItems[index] = 'Grape';
  setItems(newItems);
};

return (
  <div>
    <h1>Arrays With UseState Examples</h1>
    <ul>
      {items.map((item, index) => (
        <li key={index}>
          {item}
          <button onClick={() => removeItem(index)}>Remove</button>
          <button onClick={() => updateItem(index)}>Update</button>
        </li>
      ))}
    </ul>
    <button onClick={addItem}>Add Item</button>
  </div>
);
```

useState Example

4. Objects with useState

```jsx
import React, { useState } from 'react';

export default function ObjectsWithUseState() {
  // Initialize state
  const [user, setUser] = useState({ name: 'John', age: 30 });

  // Function to handle name change
  const changeName = () => {
    setUser(prevUser => ({ ...prevUser, name: 'Jane' }));
  };

  // Function to handle age increment
  const incrementAge = () => {
    setUser(prevUser => ({ ...prevUser, age: prevUser.age + 1 }));
  };

  return (
    <div>
      <h1>Objects With UseState Examples</h1>
      <p>{user.name} is {user.age} years old</p>
      <button onClick={changeName}>Change Name</button>
      <button onClick={incrementAge}>Increment Age</button>
    </div>
  );
}
```

useEffect

- Performs side effects in functional components
- It executes after every render cycle.

Common Use Cases for useEffect

- Fetching Data: You can use useEffect to fetch data from an API when the component mounts or when certain conditions are met.
- Setting up Subscriptions or Listeners: Useful for adding event listeners, WebSocket connections, or any form of subscription.
- Manual DOM Manipulations: For tasks that require direct manipulation of the DOM, which is typically handled outside of React's virtual DOM.
- Timers and Intervals: Setting up and clearing timers or intervals.
- Updating Document Title or Other Side Effects: Any task that needs to run in response to React rendering, but isn’t directly related to the DOM output.

useEffect Contd.

Empty Dependencies Array

When the dependencies array is empty, [], the effect will only run once. useEffect has two parameters one is the callback function and the other is the dependency array. When browser renders and useEffect is called 1st it will the run the 1st parameter which is the callback function then check for the dependency array.

```
useEffect(() => {
  console.log('This will only run once after the initial render');
}, []);
```

No Dependencies Array

- No dependencies array is provided, the effect will run after every render.
- Performing an action on every render or state/props change.

```
useEffect(() => {
  console.log('This will run after every render');
});
```

useEffect Contd.

Dependencies Array with Values

- When specific values are provided in the dependencies array, the effect will only run when any of those values change between renders.
- Fetching data when a piece of state or prop changes.
- Managing side effects conditionally, based on state/props changes.

```
const [count, setCount] = useState(0);

useEffect(() => {
  console.log('This will run when "count" state changes');
}, [count]);
```

Where to use?

1. **Empty Dependencies Array ([])**

- Fetching Data on Component Mount
- Setting Up Subscriptions Creation(Happens only one time)
- Initializing Third-Party Libraries(like a chart or map library)

1. **No Dependencies Array**

- Tracking User Interaction: Log user interactions or UI events
- Dynamic Style Changes
- Live Validations

1. **Dependencies Array with Specific Values**

- Data Fetching Based on User's Selection
  - Eg: user selects a category from a dropdown menu
- Reacting to Prop Changes in a Component
  - A DateRangePicker component allows users to select a date range. A StockChart component displays stock data for the selected range.

JavaScript Promises

As the complexity of asynchronous operations grows (like multiple asynchronous tasks that depend on each other), you end up with nested callbacks. This is often referred to as "callback hell" or "pyramid of doom."

- A Promise is an object representing the eventual completion or failure of an asynchronous operation.
- It provides .then() for success scenarios, .catch() for handling errors, and .finally() for cleanup, making the flow more readable and easier to manage.

JavaScript Promises

```
asyncOperation1((result1) => {
  asyncOperation2(result1, (result2) => {
    asyncOperation3(result2, (result3) => {
      // ... and so on
    });
  });
});
```

```
// Using the Promise
myAsyncFunction()
  .then((result) => {
    console.log(result);
  })
  .catch((error) => {
    console.error(error);
  })
  .finally(() => {
    console.log("Cleanup or finalization");
  });
```

useEffect Example

Empty Dependencies Array

```
useEffect(() => {
  fetch('https://jsonplaceholder.typicode.com/posts')
    .then(response => response.json())
    .then(data => {
      setPosts(data);
      setLoading(false);
    })
    .catch(error => {
      console.error("Error fetching data: ", error);
      setLoading(false);
    });
}, []);
```

No Dependency Array

```
// Effect runs after every render
useEffect(() => {
    console.log('Component updated! Current count:', count);
}); // No dependency array
```

useEffect Example

Dependencies Array with Values

```
useEffect(() => {
    fetch(`https://jsonplaceholder.typicode.com/users/${userId}`)
        .then(response => response.json())
        .then(data => setUserData(data))
        .catch(error => console.error('Error:', error));
}, [userId]);
```

useRef

Purpose: useRef is a hook in React that allows you to persist values across renders without triggering a re-render of the component. It's often used for accessing DOM elements directly, but it can also be used to keep a mutable value that doesn't cause a re-render when it changes.

Usage: You create a ref using useRef, and the ref's **.current** property can be used to access and modify the referenced value, such as a DOM element.

Real World Example: Animating a Component

You want to animate a component smoothly, based on some user interaction or other changes.

Using useState Problem:

Using useState for this would cause the component to re-render every time the state changes, which can be inefficient and lead to performance issues for animations.

Solution with useRef:

Instead, you can use useRef to store the current animation state. Since updating a ref does not trigger a re-render, it's more efficient for this use case.

useRef - Example

```jsx
import React, { useEffect, useRef } from 'react';

export default function UseRef() {
  const inputRef = useRef(null); // Create a ref for the input element

  useEffect(() => {
    // Focus the input element when the component mounts
    if (inputRef.current) {
      inputRef.current.focus();
    }
  }, []);

  return (
    <div>
      <input ref={inputRef} type="text" placeholder="I'll be focused on load" />
    </div>
  );
}
```

This approach is commonly used in forms, search bars, or login screens where you want to improve the user experience by reducing the need for additional clicks or taps to start typing.

useReducer

Purpose: useReducer is used to manage complex state logic in a more organized and predictable way, especially when you have state transitions that depend on the previous state.

Usage:

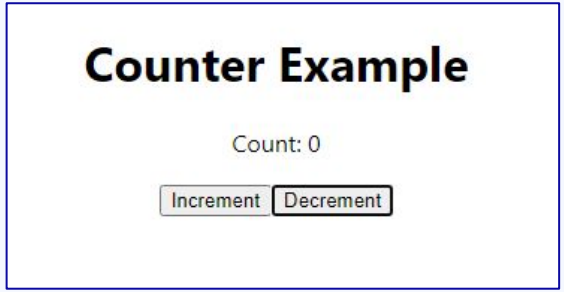
- An e-commerce shopping cart can involve various actions like adding items, removing items, applying discount codes
- Undo/Redo Functionality
- In games or any application with complex state that can change in numerous ways based on user interaction
- Complex Animations and Timers

useReducer

- **initialState:** This is indeed the first state of the application or component.
- **Reducer:** This is a function that defines how the state should change in response to an action.
- **State:** This is the current state of your application or component, as determined by the reducer.
- **Dispatch:** This is a function that you call to send actions to the reducer. When you want to update the state, you dispatch an action.

useReducer - Example

```jsx
import React, { useReducer } from 'react';

const reducer = (state, action) => {
    switch (action.type) {
        case 'INCREMENT':
          return { count: state.count + 1 };
        case 'DECREMENT':
          return { count: state.count - 1 };
        default:
          return state;
      }
};

const initialState = { count: 0 };

const Counter = () => {
  const [state, dispatch] = useReducer(reducer, initialState);

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'INCREMENT' })}>Increment</button>
      <button onClick={() => dispatch({ type: 'DECREMENT' })}>Decrement</button>
    </div>
  );
};

export default Counter;
```



useCallback

Purpose: useCallback is a hook in React that returns a memorized version of the callback function that only changes if one of the dependencies has changed. It's useful when passing callbacks to optimized child components that rely on reference equality to prevent unnecessary renders.

Usage:

- When you have a list of items rendered by a child component and each item has an event handler like an onClick
- If you have a function that triggers a complex calculation or a side effect, and that function is being used as a dependency for other hooks or childs

useCallback - Example

Parent Component

```jsx
import React, { useState, useCallback } from 'react';
import ChildComponent from './ChildComponent'; // Import ChildComponent

export default function UseCallback() {
  const [count, setCount] = useState(0);

  const handleClick = useCallback(() => {
    console.log('Button clicked');
    setCount(prevCount => prevCount + 1);
  }, []);

  return (
    <div>
      <h1>UseCallback Example</h1>
      <p>Count: {count}</p>
      <ChildComponent onClick={handleClick} />
    </div>
  );
}
```

Child Component

```jsx
import React from 'react';

const ChildComponent = React.memo(({ onClick }) => {
    console.log('ChildComponent rendered');
    return <button onClick={onClick}>Click me!</button>;
});

export default ChildComponent;
```

useMemo

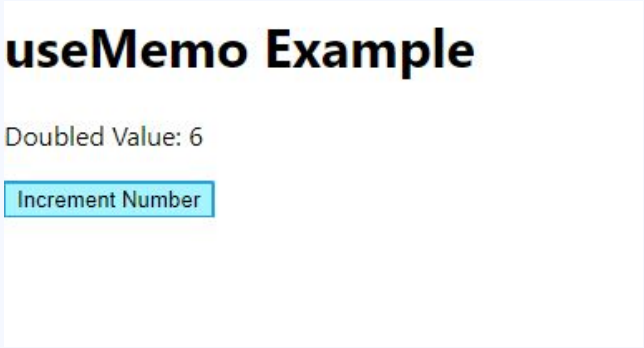
Purpose: useMemo is a React hook that memorizes values. This means it allows you to compute expensive calculations only when necessary, by remembering ("memorizing") the computed values based on a set of dependencies. It's useful for optimizing performance by avoiding expensive recalculations on every render.

Usage:

- Expensive Calculations on Each Render: Calculating aggregated data (like sums, averages, etc.)
- Applying complex filters

useMemo - Example

```jsx
import React, { useState, useMemo } from 'react';

// Simulates an expensive calculation
function expensiveCalculation(num) {
  console.log('Calculating...');
  // In a real-world scenario, this could be a more CPU-intensive task
  return num * 2;
}

export default function UseMemo() {
  const [number, setNumber] = useState(0);

  // useMemo is used to memoize the result of expensiveCalculation
  // It will only recompute the doubled value when 'number' changes
  const doubledValue = useMemo(() => expensiveCalculation(number), [number]);

  return (
    <div>
      <h1>useMemo Example</h1>
      <p>Doubled Value: {doubledValue}</p>
      <button onClick={() => setNumber(number + 1)}>Increment Number</button>
    </div>
  );
}
```



useLayoutEffect

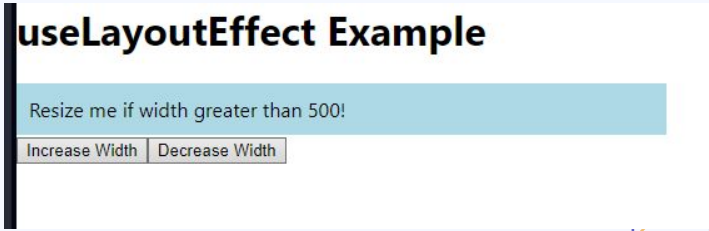
Purpose: useLayoutEffect is similar to useEffect, but it runs synchronously after all DOM mutations. This means useLayoutEffect runs right after React has made changes to the DOM, but before the browser has had a chance to "paint" those changes, making it ideal for reading layout from the DOM and re-rendering synchronously.

Usage:

- Applications like analytics dashboards, project management tools, or any customizable interface where users can interact with multiple movable or resizable elements.
- In an e-commerce app, clicking on a product image might expand it to show more detail or additional product views.

useLayout - Example

```jsx
import React, { useLayoutEffect, useRef, useState } from 'react';

export default function UseLayoutEffect() {
  const [width, setWidth] = useState(100); // Width in pixels
  const boxRef = useRef(null);

  useLayoutEffect(() => {
    if (boxRef.current) {
      const currentWidth = boxRef.current.offsetWidth;
      if (currentWidth > 500) {
        setWidth(500);
      }
    }
  }, [width]);

  return (
    <div>
      <h1>useLayoutEffect Example</h1>
      <div ref={boxRef} style={{ width: `${width}px`, backgroundColor: 'lightblue', padding: '10px' }}>
        Resize me if width greater than 500!
      </div>
      <button onClick={() => setWidth(width + 100)}>Increase Width</button>
      <button onClick={() => setWidth(width - 100)}>Decrease Width</button>
    </div>
  );
}
```



useContext

Purpose: useContext is a React hook that makes it easier to use context in your components. Context is a way to share values (like user data or theme settings) across different components in your app, without having to pass these values through each component as props.

Usage:

- Theme switcher that allows users to switch between light and dark modes.
- User authentication status (logged in/logged out) and user details need to be accessible across various components.
- For applications supporting multiple languages, you need a way to provide localized text and format data (like dates and currencies)

Provider Component

```jsx
import React, { createContext, useState } from 'react';

// Create the context
export const UseContext = createContext();

// Provider component
export const Parent = ({ children }) => {
  const [theme, setTheme] = useState('light'); // default theme is light

  // Toggle between 'light' and 'dark'
  const toggleTheme = () => {
    setTheme(prevTheme => prevTheme === 'light' ? 'dark' : 'light');
  };

  return (
    <UseContext.Provider value={{ theme, toggleTheme }}>
      {children}
    </UseContext.Provider>
  );
};
```

Child Component

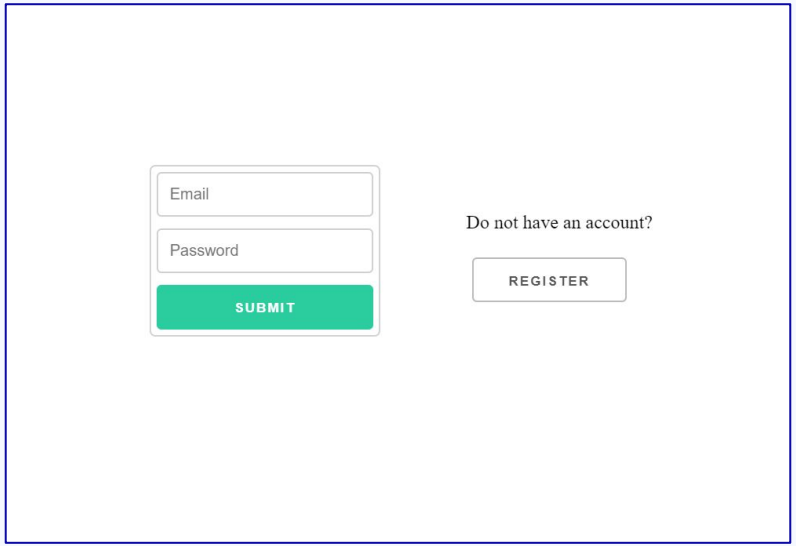
```jsx
import React, { useContext } from 'react';
import { UseContext } from './UseContext';

export default function ChildComponent() {
  const { theme, toggleTheme } = useContext(UseContext);

  return (
    <div style={{
      padding: '20px',
      backgroundColor: theme === 'light' ? '#fff' : '#000',
      color: theme === 'light' ? '#000' : '#fff'
    }}>
      <h2>Current Theme: {theme}</h2>
      <button onClick={toggleTheme}>Toggle Theme</button>
    </div>
  );
}
```

App.js

```jsx
import { Parent } from './Components/UseContext/UseContext.jsx';
import ChildComponent from './Components/UseContext/ChildComponent';
```

```jsx
<Parent>
  <ChildComponent />
</Parent>
```

Introduction to React Form Elements



Introduction to React Form Elements

React provides a way to handle form elements in two different approaches:

- **Controlled Components**
  - A Controlled Component in React is a form element (like an input) where the value of the input is controlled by React.
  - This means the value of the form element is directly tied to a value in the component's state. When the state changes, the value of the form element changes, and vice versa.
- **Uncontrolled Components**
  - An Uncontrolled Component is a form element where the value of the input is not controlled by React's state.
  - Instead of using state, useRef hook is used.

Uncontrolled Components Example

```
const firstNameRef = useRef();
const lastNameRef = useRef();

const handleSubmit = (e) => {
    e.preventDefault();
    const values = {
        firstName: firstNameRef.current.value,
        lastName: lastNameRef.current.value
    };
    console.log('Uncontrolled Form submitted with values:', values);
};
```

```
return (
    <form onSubmit={handleSubmit}>
        <input
            type="text"
            name="firstName"
            placeholder="First Name"
            ref={firstNameRef}
        />
        <input
            type="text"
            name="lastName"
            placeholder="Last Name"
            ref={lastNameRef}
        />
        <button type="submit">Submit</button>
    </form>
);
```

```
import React, { useRef } from "react";
```

The import statement brings in React and a named import useRef from the "react" library.

```
export default function Uncontrolled_Component() {
    const firstNameRef = useRef();
    const lastNameRef = useRef();
```

- The Uncontrolled_Component function is a React component defined as a default export.
- Two **useRef** hooks are used to create mutable objects firstNameRef and lastNameRef.

```
const handleSubmit = (e) => {
    e.preventDefault();
    const values = {
        firstName: firstNameRef.current.value,
        lastName: lastNameRef.current.value
    };
    console.log('Uncontrolled Form submitted with values:', values);
};
```

- The handleSubmit function is defined to handle the form submission.
- It prevents the default form submission behavior, gathers the values of the input fields using the current property of the refs, and logs them to the console.

```
    };
    return (
    <form onSubmit={handleSubmit}>
        <h1>Uncontrolled Components Example</h1>
        <input
            type="text"
            name="firstName"
            placeholder="First Name"
            ref={firstNameRef}
        />
        <input
            type="text"
            name="lastName"
            placeholder="Last Name"
            ref={lastNameRef}
        />
        <button type="submit">Submit</button>
    </form>
);
}
```

- In the return statement, a <form> element is rendered. It has an onSubmit event handler set to the handleSubmit function defined earlier.
- Inside the form, there are two <input> elements for first name and last name. Both have ref attributes set to firstNameRef and lastNameRef, respectively.
- Finally, there's a <button> element for form submission.

- When the form is submitted, the handleSubmit function is called.
- It **retrieves the values** of the input fields **directly from the DOM** nodes using the refs and logs them to the console.
- This approach is termed "uncontrolled" because React doesn't manage the form data; instead, the DOM does.
- It's useful in situations where you need to integrate with non-React code or work with form elements directly.

Controlled Components Example

1. Using separate state variables for each input:

Input values, you can either manage them with separate state variables or use a single state object.

```
const [username, setUsername] = useState("");
const [email, setEmail] = useState("");

const handleSubmit = (e) => {
  e.preventDefault();
  console.log("Username:", username);
  console.log("Email:", email);
};
```

```
return (
  <form onSubmit={handleSubmit}>
    <div>
      <label>
        Username:
        <input
          type="text"
          value={username}
          onChange={(e) => setUsername(e.target.value)}
        />
      </label>
    </div>
    <div>
      <label>
        Email:
        <input
          type="email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}
        />
      </label>
    </div>
    <button type="submit">Submit</button>
  </form>
);
```

Controlled Components Example

2. Using a single state object for all input values:

```jsx
const [values, setValues] = useState({
    firstName: "",
    lastName: ""
});

const handleChange = (e) => {
    const { name, value } = e.target;
    setValues(prevValues => ({
        ...prevValues,
        [name]: value
    }));
};

const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Controlled Form submitted with values:', values);
};
```

```jsx
return (
    <form onSubmit={handleSubmit}>
        <input
            type="text"
            name="firstName"
            value={values.firstName}
            placeholder="First Name"
            onChange={handleChange}
        />
        <input
            type="text"
            name="lastName"
            value={values.lastName}
            placeholder="Last Name"
            onChange={handleChange}
        />
        <button type="submit">Submit</button>
    </form>
```

```
import React, { useRef } from "react";
```

The import statement brings in React and a named import useRef from the "react" library.

```
export default function Controlled_Seperate() {
    // Separate state variables for each input
  const [username, setUsername] = useState("");
  const [email, setEmail] = useState("");
```

- The Controlled_Separate function is defined as a React component and is exported as the default export.
- Inside the component function, there are two state variables: username and email, managed using the **useState** hook.
- Each state variable represents the value of the respective input field.

```
const handleSubmit = (e) => {
  e.preventDefault();
  console.log("Username:", username);
  console.log("Email:", email);
};
```

- The handleSubmit function is defined to handle the form submission.
- It prevents the default form submission behavior, and then logs the values of username and email to the console.

```
return (
  <form onSubmit={handleSubmit}>
    <h1>Controlled Components with Seperate Variables Example</h1>
    <div>
      <label>
        Username:
        <input
          type="text"
          value={username}
          onChange={(e) => {
            setUsername(e.target.value);
            console.log("Username changed:", e.target.value);
          }}
        />
      </label>
    </div>
    <div>
      <label>
        Email:
        <input
          type="email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}

        />
      </label>
    </div>
    <button type="submit">Submit</button>
  </form>
  );
}
```

- Inside the form, there are two <input> elements, one for username and one for email.
- Both inputs have their **value attributes** set to the corresponding state variables (username and email). This makes them controlled components because their values are directly tied to React state.
- Both <input> elements have **onChange event handlers**. When the value of an input changes, the event handler updates the state variable (username or email) with the new value entered by the user.

```jsx
return (
  <form onSubmit={handleSubmit}>
    <h1>Controlled Components with Seperate Variables Example</h1>
    <div>
      <label>
        Username:
        <input
          type="text"
          value={username}
          onChange={(e) => {
            setUsername(e.target.value);
            console.log("Username changed:", e.target.value);
          }}
        />
      </label>
    </div>
    <div>
      <label>
        Email:
        <input
          type="email"
          value={email}
          onChange={(e) => setEmail(e.target.value)}

        />
      </label>
    </div>
    <button type="submit">Submit</button>
  </form>
  );
}
```

- The console.log statements inside the onChange event handlers log the new values of username and email to the console whenever they change, providing real-time feedback on user input.

- Finally, there's a <button> element for submitting the form.

State Management

In the first code segment (Uncontrolled Components), form data is handled by the DOM itself, and React doesn't manage the form data directly. Instead, refs are used to access the DOM nodes directly.

In the second code segment (Controlled Components), form data is controlled by React state. State variables (username and email) are used to track the values of the input fields, and updates to these values are handled through state setters (setUsername and setEmail).

Validation and Manipulation

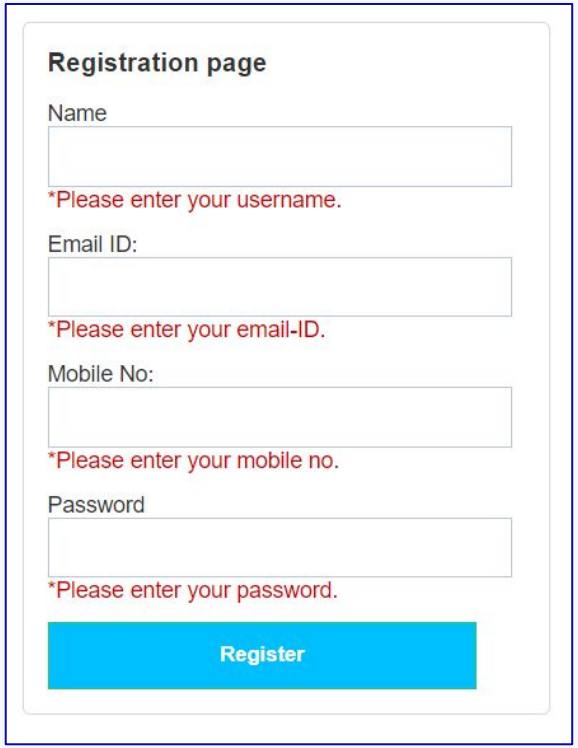
Uncontrolled components (first code segment) may require additional effort for validation and manipulation since the form data is not directly controlled by React.

Controlled components (second code segment) allow for easier validation and manipulation of form data because all changes to the input values are routed through React state.

Where to use?

- Controlled Components
  - **Dynamic Form Handling**: When the behavior of one input depends on the value/selection of another input.
  - **Instant Field Validation**: When you need to validate fields in real-time, as the user types.
- Uncontrolled Components
  - **Integration with Non-React Code**: If you're integrating with third-party libraries or components not built in React, uncontrolled components might be more suitable.
  - **File Inputs:** For inputs like the file picker, which are read-only (users can't type into them), uncontrolled components are often more straightforward.

Introduction to Form Validation



Introduction to Form Validation

Ensure that the user input matches the required format to maintain Data consistency, security, and improved user experience.

Basic Form Validation:

1. Required Fields - Check whether essential fields have input before submitting.

```
<input type="text" name="username" required />
```

2. **type** attribute - Using specific input types can also add inherent validation

```
<input type="email" name="email" />
<input type="date" name="birthdate" />
<input type="password" name="password"/>
<input type="url" name="website" />
```

Introduction to Form Validation

3. **min** and **max** attribute - For input types number, range, date, datetime-local, month, time, and week, you can set boundaries on the values.

<input type="number" name="age" min="1" max="100" />

4. **minlength** and **maxlength** attribute - Set the minimum and maximum length for input fields.

<input type="password" name="password" minlength="8" maxlength="20" />

2. Pattern check - Allows to specify a regular expression that the input field's value is checked against.

<input type="tel" name="phone" pattern="^[0-9]{10}$" title="Please enter a valid 10-digit phone number without spaces or dashes." />

Introduction to Form Validation

6. Custom Validation - You can have customized validation according to your wish, below is a customized validation to check password and confirm password when typing

```
const [password, setPassword] = useState('');
const [confirmPassword, setConfirmPassword] = useState('');
const [isMatching, setIsMatching] = useState(true);

const handlePasswordChange = (e) => {
    setPassword(e.target.value);
};

const handleConfirmPasswordChange = (e) => {
    setConfirmPassword(e.target.value);
    setIsMatching(e.target.value === password);
};
```

```
return (
    <div>
        <input
            type="password"
            placeholder="Password"
            value={password}
            onChange={handlePasswordChange}
        />
        <input
            type="password"
            placeholder="Confirm Password"
            value={confirmPassword}
            onChange={handleConfirmPasswordChange}
        />
        {!isMatching && <span style={{ color: 'red' }}>Passwords do not match</span>}
    </div>
);
```

Asynchronous validation

Let's assume we have a registration form where we want to check if the entered username is already taken. We can simulate this with a mock function, but in a real-world scenario, you'd make an API call.

Manually going to check if a username is exists or not. function will "reject" if the input matches 'john', 'alice', or 'bob'. Any other input will "resolve". The setTimeout function is used to introduce a delay.

```
const [username, setUsername] = useState("");
const [isLoading, setIsLoading] = useState(false);
const [error, setError] = useState(null);
```

```
const checkUsernameAvailability = (username) => {
  return new Promise((resolve, reject) => {
    setTimeout(() => {
      if (["john", "alice", "bob"].includes(username)) {
        reject("Username is taken");
      } else {
        resolve("Username is available");
      }
    }, 1000);
  });
};
```

Asynchronous validation

- Whenever the username state changes, this useEffect runs.
- If the username is empty, the validation won't proceed.
- Before starting the async check, it sets isLoading to true and clears any previous errors.
- After the async check is done, it will either set an error (if the username is taken) or clear the error (if the username is available). Finally, it sets isLoading to false.

```
useEffect(() => {
  if (!username) return; // Don't check if username is empty

  setIsLoading(true);
  setError(null);

  checkUsernameAvailability(username)
    .then(() => setError(null))
    .catch((err) => setError(err))
    .finally(() => setIsLoading(false));
}, [username]);

const handleSubmit = (e) => {
  e.preventDefault();
  if (!error && !isLoading) {
    console.log("Form submitted with username:", username);
  }
};
```

Asynchronous validation

- Conditional rendering is used to show a "Loading..." message or an error message based on the isLoading and error states.

```
return (
  <form onSubmit={handleSubmit}>
    <div>
      <label>Username:</label>
      <input value={username} onChange={(e) => setUsername(e.target.value)} />
      {isLoading && <span>Loading...</span>}
      {error && <span style={{ color: "red" }}>{error}</span>}
    </div>
    <button type="submit" disabled={error || isLoading}>
      Register
    </button>
  </form>
);
```

Using React Libraries for validation

Formik is a popular library that helps manage form state and validations in React.

1. Need to install these packages first:
   npm install formik yup
2. Import Necessary Libraries

```
import React from 'react';
import { useFormik } from 'formik';
import * as Yup from 'yup';
```

- useFormik which is a custom hook provided by Formik to help manage form state, handle validation, and submission.
- Yup which is a JavaScript schema builder for value parsing and validation.

Using React Libraries for validation

3. Define the Formik Configuration

- Set the initial state for the form fields.

```
const formik = useFormik({
  initialValues: {
    username: '',
    email: '',
    phoneNumber: '',
    password: '',
    confirmPassword: '',
    dob: '',
  },
```

Using React Libraries for validation

- Then define validation rules using Yup. It allows for easy-to-write and easy-to-read validations. Function will be executed when the form is submitted and is valid by onSubmit function.

```
validationSchema: Yup.object({
  username: Yup.string().required('Required'),
  email: Yup.string().email('Invalid email address').required('Required'),
  phoneNumber: Yup.string()
    .matches(/^[0-9]{10}$/, 'Enter a valid 10-digit phone number')
    .required('Required'),
  password: Yup.string().required('Required'),
  confirmPassword: Yup.string()
    .oneOf([Yup.ref('password'), null], 'Passwords must match')
    .required('Required'),
  dob: Yup.date()
    .max(new Date(), 'Date cannot be in the future')
    .min(new Date(new Date().setFullYear(new Date().getFullYear() - 120)), 'You cannot be more than 120 ye
}),
onSubmit: (values) => {
  console.log(values);
},
});
```

Using React Libraries for validation

4. Build the Form Component

```
return (
  <form onSubmit={formik.handleSubmit}>
```

5. Define Form Fields with Formik's getFieldProps

```
<input type="text" name="username" {...formik.getFieldProps('username')} />
```

No need of onChange, onBlur, and value props for each input

6. Display Validation Errors

```
{formik.touched.phoneNumber && formik.errors.phoneNumber ? (
  <div style={{ color: 'red' }}>{formik.errors.phoneNumber}</div>
) : null}
```

formik.touched.phoneNumber: Checks if the phoneNumber field has been touched

formik.errors.phoneNumber: Checks if there are any validation errors associated with the phoneNumber field.

ROUTING IN REACT

Routing in React

Routing in reactJS is the mechanism by which we navigate through a website or web-application.

- React Router is a standard library for routing in React.
- Enables the navigation among views of various components in a React Application, allows changing the browser URL, and keeps UI in sync with the URL.
- Routing allows faster navigation between views, does not require a full page reload and it improves user experience.

Client side Routing vs Server side Routing

- Server-side routing what usually happens when you are entering a URL, the browser makes an HTTP request to the server then the server re-renders the HTML into the application
- Client side routing we first render the full react app from the server, but after that when you want to change pages, the browser uses the HTML5 history API to fetch the page that has already been loaded

React Router

Client-Side routing or navigation in react applications done through using React Routers.

- Installation of react router done by
  - **npm install react-router-dom**
- Basic Components of React Router
  - BrowserRouter: Uses the HTML5 history API to keep UI in sync with the URL.
  - Route: Renders UI when a location matches the route's path.
  - NavLink: Provides navigation links.

```
<BrowserRouter>
  <Route path="/home" component={Home} />
  <Route path="/about" component={About} />
</BrowserRouter>
```

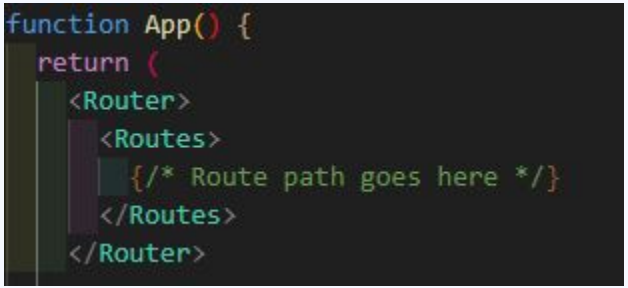
React Router : Components

1. **BrowserRouter:**
   - Wraps around the entire application or part of it to provide the capability of client-side routing. It uses the HTML5 History API to synchronize your UI with the current URL.

```
import { BrowserRouter as Router } from 'react-router-dom';

function App() {
  return (
    <Router>
    {/* other router components go here */}
    </Router>
    );
  }

  export default App;
```

React Router : Components

2. **Routes:**
    - A container for one or more Route components.
    - Routes ensures that only one of its children (Route or Redirect) matches the current location. As soon as it finds a match, it stops looking at the rest of its children
    - With Routes, nested route configurations are more straightforward and more intuitive.



React Router : Components

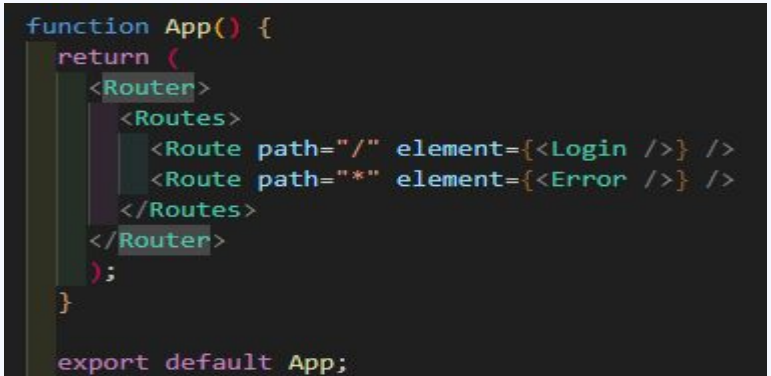
3. Route:

- It is used to define individual routes in your application. It renders the UI when its path matches the current location.
- When the location matches the path you provide, it renders the element.

```
function App() {
  return (
    <Router>
      <Routes>
        <Route path="/login" element={<Login />} />
      </Routes>
    </Router>
    );
  }

  export default App;
```

Commonly used Path notations in React Router

1. **Root Path (/):**
    - Matches the root of your app, which is typically the home page or landing page.
2. **(*):**
    - Acts as a catch-all. Matches anything that hasn't been matched by previous routes. Useful for rendering "404 - Not Found" pages.



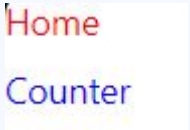
React Router : Components

4. NavLink

- Provides navigation around your application. It will create hyperlinks

```
function App() {        You, 3 weeks ago • Initialize pro
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Navigation />} />
        <Route path="/counter" element={<Counter />} />
        <Route path="*" element={<Error />} />
      </Routes>
    </Router>
    );
  }
```

```
export default function Navigation() {
    return (
        <nav>
            <NavLink to="/">Home</NavLink>
            <NavLink to="/counter">Counter</NavLink>
        </nav>
    );
}
```



React Router : Components

5. Nested Routes

- Nested Routes allow defining routes inside other routes.
- There are two ways to declare routes
  - Defining the full route path in App.js

```
<Route path="/dashboard" element={<Dashboard />} />
<Route path="/dashboard/profile" element={<Profile />} />
```

  - Defining the route path in other component

App.js

```
<Route path="/dashboard/*" element={<Dashboard />} />
```

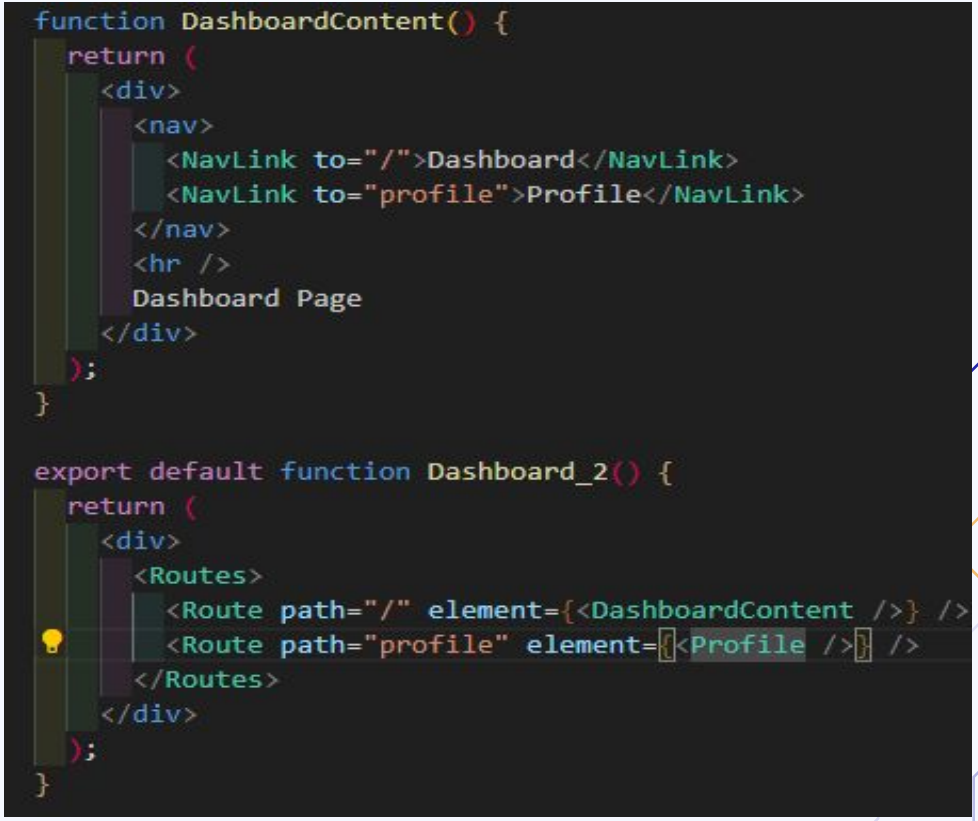
OtherComponent.jsx

```
<Routes>
  <Route path="profile" element={<Profile />} />
</Routes>
```

React Router : Components

Load as different pages

Create a separate function and call it inside the component and render.



```
function DashboardContent() {
  return (
    <div>
      <nav>
        <NavLink to="/">Dashboard</NavLink>
        <NavLink to="profile">Profile</NavLink>
      </nav>
      <hr />
      Dashboard Page
    </div>
  );
}

export default function Dashboard_2() {
  return (
    <div>
      <Routes>
        <Route path="/" element={<DashboardContent />} />
        <Route path="profile" element={<Profile />} />
      </Routes>
    </div>
  );
}
```

React Router : Components

6. Route Parameters

- Route parameters are dynamic sections of a URL, denoted with a : prefix.
- Used to capture values directly from the URL into your component.
- They allow us to access values in the URL that will be changing dynamically, often used to display the same component structure for different pieces of data.

```
<Route path="/resource/:parameter" element={<Component />} />
```

Route Parameters : Example

Setting Up a Route with Parameters in App.js

```
<Route path="/posts/:params" element={<Post />} />
```

Linking Routes in Navigation.jsx(Can be anywhere)

```
<NavLink to="/posts/article1">Article 1</NavLink>
<NavLink to="/posts/article2">Article 2</NavLink>
```

Accessing Route Parameters (Place where we are going to use it)

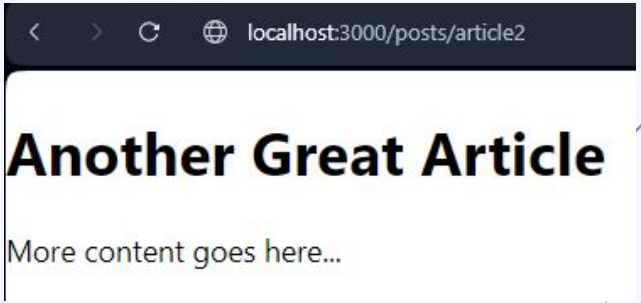
```
const { params } = useParams();
```

Route Parameters : Example

Fetching the post based on the URL parameter (Place where we are going to use it)

```
const posts = {
  "article1": {
    title: "My Awesome Article",
    content: "Content of the article goes here...",
  },
  "article2": {
    title: "Another Great Article",
    content: "More content goes here...",
  },
};

const post = posts[params];
```

```
return (
  <div>
    <h1>{post.title}</h1>
    <p>{post.content}</p>
  </div>
);
```



React Router : Components

7. useNavigate

- Primarily used when you need to navigate after some event or action (e.g., after form submission).

```
const navigate = useNavigate();
```

```
navigate('/dashboard');
```

useNavigate Example

```jsx
export default function UseNavigate() {
  const navigate = useNavigate();
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  const handleLogin = () => {
    // Simulating a login action
    setIsLoggedIn(true);

    // Navigate to another route after login
    navigate('/dashboard');
  }

  return (
    <div>
      {
        !isLoggedIn ?
        <button onClick={handleLogin}>Login</button>
        :
        <p>You are now logged in!</p>
      }
    </div>
  );
}
```

React Router : Components

8. Route Guards

- Route guards are a pattern used in routing to restrict access to certain routes based on some condition, usually user authentication
- Usage:

  Admin Panels/Dashboards: Only accessible to authenticated administrators.

  User Profiles: Where users must log in to view or edit their personal information.

  Subscription-based Services: Restricting premium content to users who have a subscription.

React Router : Components

Creating Context for Authentication : Example

```jsx
import React, { useState, createContext } from 'react';

export const AuthContext = createContext();

export const LoginContext = ({ children }) => {
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  const authenticate = () => {
    setIsLoggedIn(true);
  };

  return (
    <AuthContext.Provider value={{ isLoggedIn, authenticate }}>
      {children}
    </AuthContext.Provider>
  );
};
```

React Router : Components

Using the ProtectedRoute Component

- Wrap the routes that you want to protect with the ProtectedRoute component in App.js

```
<LoginContext>
  <Router>
    <Routes>
      <Route path="/" element={<Login />} />
      <Route element={<ProtectedRoutes />}>
        <Route path="/counter" element={<Counter />} />
      </Route>
    </Routes>
  </Router>
</LoginContext>
```

React Router : Components

Creating a ProtectedRoute Component

- This component will serve as the route guard, checking the user's authentication status before rendering the component or redirecting.
- Outlet is a component that serves as a placeholder where nested routes will be rendered.

```
import React, { useContext } from 'react';
import { Navigate,Outlet } from 'react-router-dom';
import { AuthContext } from './LoginContext';

export const ProtectedRoutes = () => {
  const { isLoggedIn } = useContext(AuthContext);

  if (!isLoggedIn) {
    return <Navigate to="/" />;
  }

  return <Outlet/>;
};
```

React Router : Components

Rendering Component

```jsx
import React, { useContext } from 'react';
import { AuthContext } from './LoginContext';
import { useNavigate } from 'react-router-dom';

export default function Login() {
    const navigate = useNavigate();
    const { authenticate } = useContext(AuthContext);

    const handleLogin = () => {
      authenticate();
      navigate("/counter");
    };

    return (
      <div>
        <h2>Login</h2>
        <button onClick={handleLogin}>Login</button>
      </div>
    );
  }
```

Thanks!